Makroverarbeitung im XCTL-System

David Damm

28th January 2004

Contents

1 Aufbau der Makrodateien

Im XCTL-System können Makros, d.h. eine Folge von verschiedenen Befehlen, verarbeitet werden. Diese Makros werden in Dateien mit der Endung `*.mak` gespeichert. Bisher gibt es zwei verschiedene Makrodateien. Die Datei `standard.mak` enthält folgende Makros: `SetupTopographie`, `SearchReflection`, `InquireHwb`, `AzimutalJustify` und `Test` (zum Testen eigener Makros). Die zweite Datei `scan.mak` enthält zwei spezielle Makros zum Durchführen von Scan's: `ScanJob` und `AreaScanJob`.

Diese beiden Makrodateien werden von dem XCTL-Programm durchforstet, und alle gefundenen Makros in eine Liste aufgenommen. Diese können dann mittels des Dialogs Ablaufsteuerung vom Nutzer ausgeführt werden.

Eine Makrodatei besteht aus Blöcken, wobei jeder Block ein Makro beschreibt. Solch ein Block setzt sich aus zwei Teilen zusammen, nämlich dem Deklarationsteil, in dem der Name und die Länge des Makros festgelegt wird, und dem Befehlsteil, in dem die nacheinander auszuführenden Befehle stehen.

Im folgenden wird der Aufbau von Makrodateien detailliert durch die angegebene Sprache beschrieben.

```
Makrodatei    = { Makro '\n' }+
Makro         = Commonblock '\n' Befehlsblock
Commonblock   = '[Common]' '\n' Namensdefinition
                           '\n' Längendefinition
Namensdefinition = 'Name' Trennzeichen Makroname
Längendefinition = 'Length' Trennzeichen Zahl
Makroname     = 'AreaScanJob' | 'AzimutalJustify' | 'InquireHwb' |
                'ScanJob' | 'SearchReflection' | 'SetupTopography' | 'Test'
Zahl          = '1' | '2' | '3' | ...
Befehlsblock  = '[Commands]' '\n' Befehlsfolge '[End]'
Befehlsfolge  = { Befehl { Trennzeichen Parameter }* { Wert }*'\n' }+
Befehl        = 'AreaScan' | 'Calculate' | 'ChooseAxis' |
                'ChooseDetector' | 'ControlFlank' | 'GotoIntensity' |
                'GotoPeak' | 'LoadPoint' | 'MoveToPoint' |
                'SaveData' | 'Scan' | 'SetFileName' |
                'SetupScan' | 'SetWidth' | 'ShowValue' | 'Stop'
Parameter     = 'LargeSide' | 'SmallSide' | 'ToSmallerAngle' |
                'Interpolation' | 'LastGoal' | 'Standard' | 'BackMove' |
                'Result' | 'Argument' | 'Hwb' | 'Difference' | 'Middle' |
                'Opposite' | 'ToLargerAngle'
```

Zusätzlich benötigen einige Befehle zusätzliche Werte, damit sie ausgeführt werden können. Dazu kann *Wert* in der *Befehlsfolge* mit unterschiedlichen Werten (Zeichenkette, Ganzzahl, Gleitkommazahl) belegt werden, wie es gerade benötigt wird (z.B. `MoveToPoint Relative 100.0` bewegt den gewählten Motor relativ zur aktuellen Position mit dem Faktor 100.0).

Weiterhin sollte man beachten, dass ein Makro stets mit dem Befehl `Stop` endet (ist das nicht der Fall, wird vom Programm intern ein `Stop` hinzugefügt).

Es ist ebenfalls wichtig, die Länge des Makros, d.h. die Anzahl der Befehle, in `Length` anzugeben. Ist die angegebene Länge zu kurz, werden alle Befehle danach ignoriert.

2 Befehle

Alle Befehle haben ein festgelegtes Format, dass sich im allgemeinen wie folgt darstellen läßt:

```
Befehl Parameter1 Parameter2 Parameter3
```

Mit `Befehl` ist der Befehlsname gemeint (siehe Sprachdefinition), also z.B. `ChooseAxis`. Jeder Befehl kann dann bis zu zwei Befehlsparameter haben, die durch `Parameter1` und `Parameter2` bestimmt werden. Der `Parameter3` dient dazu, verschiedene Werte weiterzugeben, die für die Ausführung des Befehls benötigt werden. `Parameter3` kann aus nur einem Wert bestehen (wie bei dem Befehl `MoveToPoint`) oder aus mehreren Werten, also einer Liste (wie bei `SetupScan`). Es besteht auch die Möglichkeit, dass der dritte Parameter nicht benötigt wird (z.B. bei `ChooseAxis`).

In den folgenden Abschnitten wird auf die einzelnen Befehle genauer eingegangen.

2.1 AreaScan

Der Befehl `AreaScan` besitzt zwei Programmparameter:

```
AreaScan { ForScan | ForAreaScan ] [ StandardScan | Omega2ThetaScan } .
```

2.2 Calculate

Der Befehl `Calculate` besitzt nur einen Programmparameter. Die Syntax lautet:

```
Calculate { Difference | Opposite | Hwb | Middle } .
```

Der Befehl dient dazu, vorher mittels `LoadPoint` geladene Werte miteinander zu verrechnen. Das Ergebnis wird dann in der Variablen `Result` gespeichert, außer bei der Berechnung der Halbwertsbreite, wird das Ergebnis in `Hwb` gespeichert. Die berechneten Werte können dann zum Beispiel mit `ShowValue` angezeigt werden.

Je nachdem, welcher Parameter angegeben wurde, wird eine der folgenden Berechnungen durchgeführt.

1. `Difference`

   Hier wird die Differenz zwischen zwei Werten berechnet, die mit `LoadPoint` zwischengespeichert wurden. Das Ergebnis wird dann nach der Formel

   ```
   Result = Argument2 - Argument1
   ```

   berechnet.

2. `Opposite`

   Bei Angabe dieses Parameters, wird der Peak hergenommen und das erste Argument davon abgezogen.

   ```
   Result = 2 * PeakPoint - Argument1
   ```

3. `Hwb`

   Es wird die Halbwertsbreite ermittelt. Die Halbwertsbreite wird zunächst wie bei `Difference` berechnet.

```
Hwb = Argument2 - Argument1
```

Ist die geforderte Einheit in Grad oder Minuten angegeben, so erfolgt noch eine zusätzliche Umrechnung der Halbwertsbreite.

```
Hwb = 60 * Hwb
```

4. `Middle`

   Hier wird der arithmetische Mittelwert von zwei Werten bestimmt.

   ```
   Result = ( Argument1 + Argument2 ) / 2
   ```

2.3 ChooseAxis

Der Befehl `ChooseAxis` hat genau einen Programmparameter, der den auszuwählenden Antrieb beschreibt:

```
ChooseAxis { DC | DF | CC | AR | TL | Omega | Phi |
             Psi | Theta | X | Y | Z | Absorber } .
```

Die Bezeichnung der Antriebe hängt vom Versuchsaufbau ab. In der Topographie werden `DF`, `DC`, `AR`, `TL` und `CC` verwendet. In der Diffraktometrie/Reflektometrie verwendet man `Omega`, `Phi`, `Psi`, `X`, `Y`, `Z` sowie `CC`, `Theta` und `Absorber`.

Es gibt einige Antriebe die Synonym verwendet werden. Folgende Antriebe haben die gleiche Bedeutung `AR == Phi`, `DF == Omega` und `TL == Psi`.

2.4 ChooseDetector

Der Befehl `ChooseDetector` wählt einen aktuellen Detektor aus:

```
ChooseDetector { Counter | PSD } [ Time Counts Fl ] .
```

Die Parameter `Time`, `Counts` und `Fl` sind optional und dienen zur Einstellung spezieller Optionen des Detektors.

2.5 ControlFlank

Dieser Befehl wird benutzt, um eine Flanke zu finden bzw. zu kontrollieren, ob sich an der aktuellen Motorposition eine Flanke befindet:

```
ControlFlank { SmallSide | LargeSide } ControlRange .
```

Für `ControlRange` muß eine Zahl angegeben werden. Diese Zahl gibt den Bereich der Flanke an, d.h. von der aktuellen Intensität aus, wird eine minimale Intensität $I_{min} = \text{fIntensity}[0] \cdot (1 - \text{ControlRange}/2)$, eine maximale Intensität $I_{max} = \text{fIntensity}[0] \cdot (1 + \text{ControlRange}/2)$ und daraus der Intensitätsbereich $\text{IntensityRange} = (I_{max} - I_{min})/2$ berechnet.

Die Ausführung des Befehls ist schwer durchschaubar. Es ist auch nicht klar, wie nachgeregelt (was ist genau gemeint?) wird.

2.6 GotoIntensity

Mit dem Befehl `GotoIntensity` kann man eine bestimmte Zielintensität relativ zur Peakintensität anfahren:

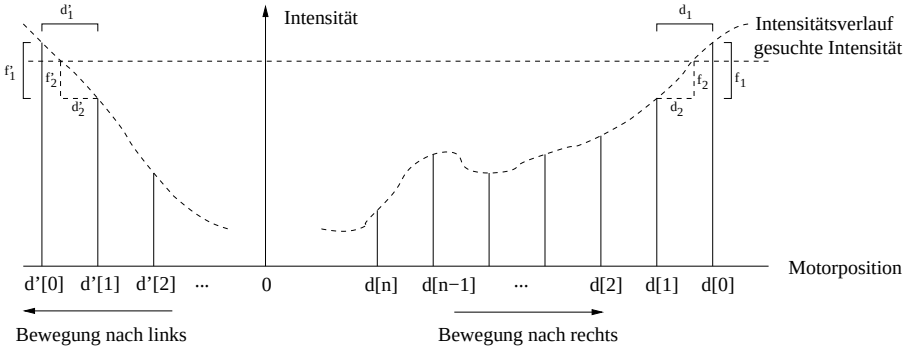
```
GotoIntensity { SmallSide | LargeSide } { Interpolation | BackMove } Value .
```

Der erste Programmparameter gibt die Bewegungsrichtung an, entweder nach links (`SmallSide`) oder nach rechts (`LargeSide`).

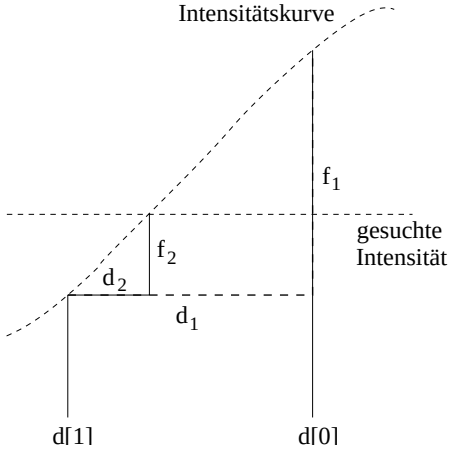
Im zweiten Parameter wird angegeben, was passieren soll, wenn die Zielintensität erreicht wurde. Bei `BackMove` wird der Motor um einen Schritt zurückgefahren. Bei der Interpolation wird die anzufahrende Position aus den letzten beiden Positionen ermittelt.
Da der Antrieb aber stets mit der angegebenen Schrittweite (`SetWidth`) bewegt wird, bringt diese Interpolation (linear) nichts, weil die berechnete Position zwischen zwei Punkten liegt, die nur einen Schritt auseinander liegen. Man sollte also vorher die Schrittweite verkleinern, damit der berechnete Punkt erreicht werden kann.

Im dritten Parameter `Value` wird ein Wert angegeben, mit dem die Zielintensität berechnet wird: `Zielintensität = Value * Peakintensität`.
Liegt der Wert nun aber nicht zwischen 0 und 1, so kann die gewünschte Intensität nie erreicht werden. Dieses sollte im Programm abgefangen werden.

Die Funktionsweise von `GotoIntensity` wird in den folgenden Abbildungen verdeutlicht.



Die Abbildung zeigt einmal eine Motorbewegung nach rechts (`LargeSide`) und einmal nach links (`SmallSide`). Für die lineare Interpolation der Position zur gewünschten Zielintensität wird der Strahlensatz angewendet (siehe folgende Abbildung).



Nach dem Strahlensatz gilt dann

$$\begin{aligned} \frac{d_1}{f_1} &= \frac{d_2}{f_2} \\ \Rightarrow \quad d_2 &= \frac{d_1 \cdot f_2}{f_1} \\ &= \frac{(d\,[0] - d\,[1]) \cdot (\text{fSearchIntensity} - f\,[1])}{f\,[0] - f\,[1]}. \end{aligned}$$

Betrachten wir das konkrete Beispiel: sei $d\,[1] = 10$, $d\,[0] = 11$, $f\,[1] = 900$, $f\,[0] = 1200$ und die gesuchte Intensität fIntensitySearch $= 1000$. Dann erhalten wir

$$d_2 = \frac{(11 - 10) \cdot (1000 - 900)}{1200 - 900} = \frac{1}{3}.$$

Die neue Position ergibt sich dann als

$$\text{pos} = d\,[1] + d_2 = 10 + \frac{1}{3}.$$

Analog gilt dies auch für `SmallSide`.

Der geänderte Quellcode befindet sich in Kapitel 3.

Es ist noch anzumerken, dass zwar alle Positionswerte und zugehörigen Intensitätswerte gespeichert werden, aber immer nur die jeweils letzten beiden für Berechnungen verwendet werden. Dadurch könnte man das unnötige Shiften im Speicher (LIFO) einsparen und die Übersichtlichkeit des Programms erhöhen.

2.7 GotoPeak

Mit diesem Befehl läßt sich ein Peak bestimmen:

```
GotoPeak { ToSmallerAngle | ToLargerAngle } { Interpolation | BackMove} .
```

Dieser Befehl funktioniert ähnlich wie `GotoIntensity`, nur das hier erst noch nach dem Peak gesucht werden muß.

2.8 LoadPoint

Für den Befehl `LoadPoint` gibt es zwei verschiedene Aufrufmöglichkeiten, die erste lautet:

```
LoadPoint { Start | Peak } .
```

Hierbei wird die zuletzt erfolgreich angefahrene Position (sogenanntes `Goal`) entweder in `Start` oder in `Peak` gespeichert. `Start` soll dazu dienen, die ursprüngliche Startposition eines Antriebes vor einer Bewegung zu speichern, um möglicherweise wieder dorthin zurückzukehren. `Peak` ist dazu gedacht, einen zweiten Wert (z.B. den mittels `GotoPeak` ermittelten Abstand zur absoluten Null) zu speichern.

Die zweite Variante wird verwendet, um Werte in den "Rechner" zu laden und diese dann mit `Calculate` zu verarbeiten:

```
LoadPoint Argument Value .
```

Der Programmparameter `Argument` gibt an, dass die zuletzt erfolgreich angefahrene Position in `Argument1` oder `Argument2` ... (siehe `Calculate`) gespeichert werden soll. `Value` gibt die Position an. So bewirkt der Aufruf `LoadPoint Argument 2`, dass die zuletzt erreichte Position (`Goal`) in `Argument2` gespeichert wird.

Der Befehlsname `LoadPoint` ist irreführend, da ja nichts geladen, sondern wohl eher gespeichert wird. Vielleicht sollte die Bezeichnung in `SavePoint` oder `SetPoint` umgeändert werden.

2.9 MoveToPoint

Mit dem Befehl `MoveToPoint` kann der aktuelle Antrieb bewegt werden:

```
MoveToPoint { Result | Peak | Start | LastGoal } .
```

Damit wird der Antrieb auf verschiedene Positionen gefahren. Ist der Programmparameter `Result` angegeben, so wird der Motor zu der mittels `Calculate` berechneten Position gefahren. Bei `Peak` und `Start` fährt der Antrieb zu den dort angegebenen Positionen (sofern diese zuvor belegt worden sind). Wird LastGoal als Parameter angegeben, so wird zur letzten Position zurückgefahren (das ist insbesondere nützlich, wenn der Antrieb eine Software- oder Hardwareschranke erreicht, so kann er nämlich wieder zur letzten gültigen Position gefahren werden).

Eine weitere Möglichkeit wird durch das relative Ansteuern einer Position geboten. Dabei wird die unter `Value` angegebene Position zur aktuellen Position hinzuaddiert:

```
MoveToPoint Relative Value .
```

Zuletzt ist es auch möglich, eine absolute Position anzufahren. Diese wird einfach hinter dem Befehlsnamen (also in `Value`) angegeben:

```
MoveToPoint Value .
```

Der Aufruf `MoveToPoint 0` führt dazu, dass der Antrieb zur absoluten Null bewegt wird.

2.10 SaveData

Mit diesem Befehl werden berechnete Ergebnisse eines Scans in eine zuvor erzeugte Datei (`SetFileName`) gespeichert:

```
SaveData { ForScan | ForAreaScan } .
```

Der Parameter gibt an, welche Daten gespeichert werden sollen, nämlich die des Scans oder des AreaScans.

2.11 Scan

Damit führt man einen Scan durch:

```
Scan { ForScan | ForAreaScan } .
```

2.12 SetFileName

Dieser Befehl setzt den Namen für eine Ausgabedatei (für Scans):

```
SetFileName Name .
```

2.13 SetupScan

Damit werden für den Scan Eintellungen vorgenommen:

```
SetupScan { StandardScan | Omega2ThetaScan } ScanMin ScanWidth ScanMax .
```

Die drei letzten Parameter geben den Startpunkt des Scans, die Schrittweite und den Endpunkt des Scans an.

2.14 SetWidth

Dem Befehl `SetWidth` muß ein Parameter übergeben werden, der die festzulegende Schrittweite enthält:

```
SetWidth Value .
```

Der Wert `Value` kann eine beliebige Zahl enthalten, also zum Beispiel `Value = 12` oder `Value = 0.034`.

Über die Einheit des anzugebenden Wertes sind keine Angaben gemacht (vielleicht in Encoderschritten, dann dürften aber nur positive ganze Zahlen als Wert übergeben werden).

2.15 ShowValue

Der Befehl ShowValue dient dazu, dem Nutzer die Werte der Halbwertsbreite, des Startpunktes oder des Peaks anzuzeigen:

```
ShowValue { Hwb | Start | Peak } .
```

Die Halbwertsbreite wird in Bogensekunden ausgegeben. Beim Peak wird neben der Position zusättzlich die Intensität des Peaks ausgegeben.

3 Quellcode

3.1 GotoIntensity

```
TGotoIntensityCmd::TGotoIntensityCmd ( TCmdTag ct ) :  TCmd(ct), dDistance(0), fIntensity(0)
{
        nMotor = mlGetAxis();

        //testen, ob Parameter p1 == SmallSide (nach links bewegen)
        //                oder p1 == LargeSide (nach rechts bewegen)
        bSmallAngleSide = (ct.P1 == SmallSide);

        //zweiter Programmparameter gibt die Art der Bewegung an -> Interpolation oder BackMove
        nReadyAction = (TCParam)ct.P2;

        //Zielintensität berechnen, p3 gibt Intensität relativ zur PeakIntensity an
        fSearchIntensity = Steering.fPeakIntensity * atof(ct.P3);

        //ist gesuchte Intensität null, so war p3 == 0 oder PeakIntensity == 0 -> Fehler
        //was passiert aber, wenn p3 > 1 und PeakIntensity != 0
        //  -> dann ist die Zielintensität größer als die Intensität vom Peak,
        //     der aber die maximale Intensität hat -> also kann die Zielintensität nie erreicht werden
        if (!fSearchIntensity)
        {
                nFailureId = 11;
                eStep = CReady;
                return ;
        }

        dDistance  = new double [nLifo];
        fIntensity = new float  [nLifo];

        //bewege Motor um die doppelte Schrittweite nach links
        if (bSmallAngleSide)
                StartMove(nMotor, mGetValue(Distance) - 2.0 * mGetValue(Width));
        //bewege Motor um die doppelte Schrittweite nach rechts
        else
                StartMove(nMotor, mGetValue(Distance) + 2.0 * mGetValue(Width));

        eStep = CFirstStep;
};
```

```
TCCode TGotoIntensityCmd::FirstStep ( void )
{
        eStep = CControlStep;

        //aktuelle Intensität und Position erfassen
        fIntensity[0] = Steering.GetIntensity();
        dDistance[0]  = Steering.GetDistance();

        //erneut um einen Schritt bewegen -> warum?
        //hätte doch auch in obiger Methode erledigt werden können oder
        //die Bewegung von oben wird hierher verlagert, d.h.  dann eine Bewegung um 3 Schritte
        if (bSmallAngleSide)
                StartMove(nMotor, dDistance[0] - mGetValue(Width));
        else
                StartMove(nMotor, dDistance[0] + mGetValue(Width));
        return CRecall;
};


TCCode TGotoIntensityCmd::ControlStep ( void )
{
        double dDist;
        float idif0, idif1;

        //alle Intensitäts- und Positionswerte um eins nach rechts verschieben
        memmove((LPSTR)(fIntensity + 1), (LPSTR)fIntensity, (nLifo - 1) * sizeof(float));
        memmove((LPSTR)(dDistance  + 1), (LPSTR)dDistance,  (nLifo - 1) * sizeof(double));

        //aktuelle Intensitäts- und Positionswerte bestimmen
        fIntensity[0] = Steering.GetIntensity();
        dDistance[0]  = Steering.GetDistance();

        //Abstand der aktuellen Intensität zur Zielintensität
        idif0 = fSearchIntensity - fIntensity[0];

        //Intensitätsabstand vor einem Schritt
        idif1 = fSearchIntensity - fIntensity[1];

        //Überprüfen, ob sich beide Differenzen im Vorzeichen unterscheiden
        //falls ja, so befindet sich die gesuchte Intensität zwischen den letzten beiden,
        //also zwischen fIntensity[0] und fIntensity[1]
        if ((idif0 * idif1) < 0.0)
```

```
{
        eStep = CReadyStep;

        //Berechnen der endgültigen Position und Anfahren
        switch (nReadyAction)
        {
                //interpoliere die endgültige Position mit Hilfe der letzten beiden Positionen
                //Berechnung erfolgt unter Ausnutzung des Strahlensatzes
                case Interpolation:
                        dDist  = (fSearchIntensity - fIntensity[1]) * (dDistance[0] - dDistance[1]);
                        dDist /= (fIntensity[0] - fIntensity[1]);
                        StartMove(nMotor, dDist);
                break;

                //fahre zurück zur Position, bevor die gesuchte Intensität erreicht wurde
                case BackMove:
                        StartMove(nMotor, dDistance[1]);
                break;
        }
        return CRecall;
}

//#Hier wird nach links/rechts von der Flanke unterschieden und einmal
//#der aktuelle Intensitaetsabstand (idif0) betrachtet (links von der Flanke)
//#und das andere Mal der Intensitaetsabstand eine Runde zuvor (idif1) (rechts
//#von der Flanke)-> Hier liegt ein Fehler vor!

//es ist nicht klar, warum hier extra noch nach idif0 und idif1 unterschieden wird
//genügt es nicht, nur auf SmallAngle oder LargeAngle zu testen?
if (bSmallAngleSide)
{
        if (idif0 > 0.0)
                StartMove(nMotor, dDistance[0] + mGetValue(Width));
        else
                StartMove(nMotor, dDistance[0] - mGetValue(Width));
}
else
{
        if (idif1 > 0)
                StartMove(nMotor, dDistance[0] - mGetValue(Width));
        else
                StartMove(nMotor, dDistance[0] + mGetValue(Width));
```

```
        }
        return CRecall;
};


TCCode TGotoIntensityCmd::ReadyStep ( void )
{
        double mm;
        switch (nReadyAction)
        {
                case Interpolation:
                        mm = (fIntensity[0] - fIntensity[1]);
                        if (!mm)
                        {
                                Steering.dGoalDistance = dDistance[0];
                                Steering.fGoalIntensity = fIntensity[0];
                                eStep = CReady;
                                return CReady;
                        }
                        //erneute Berechnung der Interpolation und Zuweisung der Werte
                        //  -> hätte doch schon einen Schritt vorher getan werden können,
                        //     als der Motor zu dieser Stelle bewegt wurde
                        mm = (dDistance[0] - dDistance[1]) * (fSearchIntensity - fIntensity[1]) / mm;
                        Steering.dGoalDistance = dDistance[1] + mm;
                        Steering.fGoalIntensity = fSearchIntensity;
                        eStep = CReady;
                        break;

                default:  //BackMove
                        Steering.dGoalDistance = dDistance[1];
                        Steering.fGoalIntensity = fIntensity[1];
                        eStep = CReady;
                        break;
        }
        return CReady;
};
```